一种面向科学计算的数据流优化方法

申小伟[1),2)]　叶笑春[1)]　王　达[1)]　张　浩[1)]　王　飞[3)]　谭　旭[1),2)]
张志敏[1)]　范东睿[1)]　唐志敏[1)]　孙凝晖[1)]

[1)]（中国科学院计算技术研究所计算机体系结构国家重点实验室　北京　100190）
[2)]（中国科学院大学计算机与控制学院　北京　100049）
[3)]（中国科学院电子学研究所　北京　100090）

摘　要　传统数据流结构通过多上下文来隐藏指令等待源操作数的延迟，然而这种隐藏方式只能部分提高数据流处理器执行单元的利用率. 在面向例如 Stencil、FFT 和矩阵乘法等典型科学应用时，传统数据流结构的执行单元利用率仍然较低. 科学计算中的核心程序一般是对不同数据进行相同的操作，而且这些操作可以并行执行，数据之间没有直接依赖关系. 传统数据流结构是面向通用计算的，通常采用循环来实现对不同数据的相同操作. 在这些循环中，迭代是按照顺序依次执行的，这导致了传统数据流结构没有利用科学计算的并行性来提高性能. 所以传统数据流结构在处理这些规则的科学应用时没有协调好数据流计算模式和科学计算特征，而数据流计算是非常适合科学计算这种类型的规则计算. 基于科学计算的这些特征，该文提出了一种面向科学计算的数据流结构优化方法：循环流水优化方法. 循环流水优化方法利用科学计算的分块和并行处理特征，对传统数据流结构中的上下文控制逻辑进行了改进，将科学计算中的循环采用硬件自迭代的方式实现，并将上下文切换逻辑进行了流水化，使数据流结构中的上下文以流水线方式进入执行单元阵列，从而提高计算单元的利用率. 面对这种循环流水优化后的数据流结构，传统数据流结构上的指令映射算法不再适用. 通过分析循环流水优化后的结构特征，该文进一步提出了一种改进的指令映射算法：LBC（Load Balance Centric）指令映射算法. LBC 算法按照深度优先顺序依次映射数据流图中的所有指令，对每条指令分别计算执行单元阵列中所有位置的代价，取最小代价的位置作为最佳映射位置. LBC 算法以执行单元负载均衡为核心，同时将定点指令和浮点指令分开处理，保证执行单元上的定点部件和浮点部件的负载均衡. 每当映射一条指令时，LBC 算法采用相邻节点传输延迟与已经映射的该类型指令数量的乘积作为负载代价，来实现计算部件的负载均衡. 另外，LBC 算法将网络拥堵也作为指令映射的影响因素. LBC 算法将节点与所有父节点的距离之和作为传输代价，使指令间传输消息的路径最短，从而减小片上网络消息传递的跳数. 实验结果表明，在处理典型科学应用时，相比于传统数据流结构，循环流水的优化方法将数据流结构的性能平均提高了 4.6%. 相比于传统指令映射算法 SPDI 和 SPS，在循环流水优化后的数据流结构上，LBC 指令映射算法将性能分别平均提升了 182.6%和 158.1%.

关键词　指令映射；数据流；循环流水；科学计算处理器；高性能计算
中图法分类号　TP393　　**DOI 号**　10.11897/SP.J.1016.2017.02181

《计算机学报》

Optimizing Dataflow Architecture for Scientific Applications

SHEN Xiao-Wei[1),2)]　YE Xiao-Chun[1)]　WANG Da[1)]　ZHANG Hao[1)]　WANG Fei[3)]　TAN Xu[1),2)]
ZHANG Zhi-Min[1)]　FAN Dong-Rui[1)]　TANG Zhi-Min[1)]　SUN Ning-Hui[1)]

[1)]（*State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing*　100190）
[2)]（*School of Computer and Control Engineering, University of Chinese Academy of Sciences, Beijing*　100049）
[3)]（*Institute of Electronics, Chinese Academy of Sciences, Beijing*　100090）

Abstract　Traditional dataflow architectures hide the latency of waiting for operands through multiple contexts. But multiple contexts can only improve the utilization of function units in

收稿日期：2015-11-26；在线出版日期：2016-09-22. 本课题得到国家“八六三”高技术研究发展计划项目基金（2015AA01A301，2012AA010901）、国家核高基重大专项（2013ZX0102-8001-001-001）、国家自然科学基金（61332009，61173007，61204047，61221062）资助. **申小伟**，男，1987 年生，博士研究生，主要研究方向为处理器微结构、高性能计算机架构. E-mail：shenxiaowei@ict.ac.cn. **叶笑春**，男，1981 年生，副研究员，主要研究方向为高性能计算机架构、计算机软件仿真技术. **王　达**，女，1981 年生，博士，副研究员，主要研究方向为处理器微结构、超大规模集成电路设计. **张　浩**，男，1980 年生，博士，副研究员，主要研究方向为处理器微结构、高性能计算机架构. **王　飞**，男，1981 年生，博士，副研究员，主要研究方向为处理器微结构、FPGA 设计. **谭　旭**，男，1991 年生，博士研究生，主要研究方向为高性能计算机架构、计算机软件仿真技术. **张志敏**，男，1963 年生，博士，研究员，主要研究领域为 SOC 设计、计算机体系结构. **范东睿**，男，1979 年生，博士，研究员，主要研究领域为低功耗设计、处理器微结构. **唐志敏**，男，1966 年生，博士，研究员，主要研究领域为高性能计算机体系结构、数字信号处理. **孙凝晖**，男，1968 年生，博士，研究员，主要研究领域为高性能计算机系统.

processing elements of dataflow architectures partly. When dealing with typical scientific applications such as stencil, FFT and matrix multiplication, they are still not efficient enough because the utilization of function units is not very high. In the kernels of scientific applications, the same operations are usually performed on different data. Because the data are usually independent of each other, the operations on different data can be performed in parallel. Traditional dataflow architectures are general purpose. They usually implement the same operations on different data in loops where iterations are executed in sequence. It results in that traditional dataflow architectures don't exploit the parallelism of scientific applications. Dataflow computing is very suitable for scientific applications but traditional dataflow architectures don't coordinate the dataflow computing model and the features of scientific applications. Based on the computing features of scientific applications, in this paper, we propose an optimization of dataflow architectures for scientific applications: loop-in-pipeline optimization. The optimization takes advantages of the blocking and parallelism features of scientific applications and improves the context control logic of traditional dataflow architectures. The optimization implements the loops of scientific applications in hardware and switches the contexts in pipeline model. The loop-in-pipeline dataflow architecture streams the contexts into the processing element (PE) array in pipeline model to improve the utilization of function units. But the traditional dataflow instruction mapping algorithms are not adapted to the loop-in-pipeline architectures. Based on the features of the loop-in-pipeline architectures, we propose a novel instruction mapping algorithm: LBC (load-balance-centric) dataflow instruction mapping algorithm. The LBC algorithm maps the instructions into the PE array according to the depth of each instruction in the dataflow graph. For each instruction, the LBC algorithm computes the cost of each position of the PE array and takes the position of the minimum cost as the best position. The LBC algorithm focuses on load balancing. To balance the integer function units and floating point function units in each PE, the LBC algorithm maps the fixed instructions and float instructions separately. When an instruction is going to be mapped to the PE array, the LBC algorithm takes the multiplication of the transfer delay of adjacent PEs and the number of the same type (fixed of floating) instructions which has been mapped to the PE as the load-cost. Besides, the LBC algorithm augments the algorithm with the consideration of network contention. The sum of the distances between an instruction and its predecessor instructions is taken as the transfer-cost. The transfer-cost makes the path of each message transferred in the on-chip network minimum and reduce the hops of messages in the network. Experimental results show that the loop-in-pipeline optimization achieves a 4.6% average performance improvement over the traditional dataflow architectures and the LBC algorithm achieves a 182.6% and a 158.1% average performance improvement over SPDI and SPS algorithm respectively on typical scientific applications.

Keywords instruction mapping; dataflow; loop-in-pipeline; scientific processing unit; high-performance computing

1 引 言

大规模科学装置是高性能计算领域非常重要的应用，这些应用中的计算核心通常包括矩阵乘法、Stencil、快速傅里叶变换(Fast Fourier Transformation, FFT)等.而常见的高性能计算基准测试程序中主要包含矩阵乘法或矩阵向量乘法，这与实际的科学应用的计算特征不同，尤其体现在计算访存比上，实际科学应用的很多计算核心的计算访存比较低.这导致了现代高性能计算机在处理实际科学应用时的效率不高，浪费了大量的计算资源和能量.随着现代科学计算规模的激增，以及科学模拟准确度的大幅提高，现代高性能计算机已经无法满足现代大规模科

学装置的需求.随着大规模科学应用对高性能计算机性能功耗比要求的提高,计算效率成为高性能计算机的主要问题.

随着计算机体系结构的发展,领域专用的计算机体系结构成为主要发展趋势.在面向特定应用时,专用型结构利用应用特征对结构进行相应的优化,从而更好地发挥出硬件的计算性能.在高性能计算领域,数据流计算是领域专用计算结构的一个重要分支,数据流计算表现出了较好的性能和适用性[1-3].数据流指令执行的基本原则是:如果所有的源操作数已准备好,则该指令即可执行,这代表着数据流指令不需要按照程序计数器(Program Counter)的顺序进行[4].因此,数据流程序直接体现了程序内数据的相互依赖,数据流计算模式能够最大限度地挖掘程序的内在并行性.数据流计算模式中,源指令(生产者)执行的结果不会写入共享寄存器或共享缓存,而是直接传递给目的指令(消费者).数据流计算模式大量减少了共享存储的访问次数,提高了存储带宽的有效利用率.同时,数据流结构中的执行单元(Processing Element)都相对简单,执行单元内仅有少数寄存器和若干计算部件,其取指、译码等逻辑都非常简单.因此,执行单元的大部分面积都是用于计算部件,使得执行单元的能效比很高.因此,相比于传统的冯·诺依曼计算模式,数据流计算模式可以取得更高的性能表现[5].

在数据流计算模式中,程序是以数据流图来表示的.数据流计算中一个关键的问题是如何将数据流图映射到多个执行单元上[6-7].数据流指令映射需要兼顾通信延迟和并发性.在发掘程序的并行性的同时,尽量缩短指令之间的通信延迟,从而提高数据流程序的执行效率.例如在 TRIPS[8] 中,程序块被分配到 4×4 的执行单元阵列上,每个执行单元上最多可以放置 8 条指令,单个程序块最大为 128 条指令.执行单元、缓存和寄存器堆之间通过二维网格网络(MESH)进行通信.当一个程序块完成计算后,调度单元将下一个程序块映射到执行单元阵列上.考虑通信延迟,互相依赖的指令应该尽量映射到较近的执行单元或同一个执行单元上;考虑并发性,互相独立的指令应该尽量映射到不同的执行单元.指令映射对提高数据流计算性能有着重大的作用.

在科学计算中,数据通常都是非常规则的,而且科学计算通常是对不同的数据进行同样的操作.例如,Stencil[9] 是对多维数据中的每个点进行一次更新,FFT[10] 是对每一行做 FFT 运算.这些同样的操作一般是通过循环来完成.在科学计算中,数据可以通过分块进行处理,而且分块后每次循环处理一个子块,通常子块之间是没有数据依赖的,因此分块后的子块与子块之间可以并行处理.然而,传统数据流处理器是面向通用计算的,通用计算没有固定的特征,所以传统数据流处理器在结构设计时必须考虑到所有可能的情况,具有处理所有不同类型程序的功能.在处理通用应用程序时,传统数据流处理器中的功能单元利用率不高,这是由于多指令多线程的执行方式仍然无法完全掩盖计算单元中等待操作数的延迟,导致功能单元中没有指令可以执行.而科学计算是一类具有分块和并行性特征的应用程序,传统数据流结构没有利用这些特征进行性能优化.

在面向科学计算时,科学计算中的分块和并行性可以解决传统数据流结构中功能单元利用率较低的问题.通过将数据分块后以流水线的执行方式输入计算阵列,使每个计算单元以流水线的方式接收所有的计算数据,计算单元不再因为等待操作数而导致没有指令可以执行,从而提高数据流结构的功能单元利用率.因此,本文提出了一种面向科学计算的数据流结构优化方法:循环流水优化方法.循环流水优化方法利用科学计算的分块和并行处理等特征,对传统数据流结构中的上下文控制逻辑进行了改进,将科学计算中的循环采用硬件自迭代的方式实现,并将上下文切换逻辑进行了流水化,使数据流结构中的上下文以流水线方式进入执行单元阵列.循环硬件自迭代和流水化的最大优点是使执行单元具有更多可以执行的指令,从而使执行单元的利用率得到提高.

由于循环流水优化的数据流结构的流水线执行方式与传统数据流结构的执行方式不同,传统数据流结构上的指令映射方法不再适用于循环流水优化的数据流结构.因此,基于这种循环流水结构的计算特征,本文进一步提出了一种相应的指令映射算法:LBC(Load Balance Centric)数据流指令映射算法.LBC 根据循环流水优化后结构上指令执行的特征,以执行单元负载为核心,同时考虑定点浮点和网络拥堵的影响因素,将数据流指令映射到所有执行单元上,提高了循环流水优化后的数据流结构的性能.

本文第 2 节介绍传统数据流处理器结构及其指令映射算法;第 3 节主要介绍面向科学计算的数据流结构的循环流水优化方法;第 4 节介绍面向循环流水优化后的数据流结构的 LBC 指令映射算法;第 5 节介绍实验环境和实验结果,并对实验结果加以分析;最后,总结本文工作,并分析后续工作方向.

《计算机学报》

2 传统数据流处理器结构及其指令映射算法

2.1 传统数据流结构

传统数据流结构主要分为粗粒度数据流结构和细粒度数据流结构. 粗粒度数据流结构有TERAFLUX[11]、Runnemede[12]等,粗粒度数据流结构一般将应用程序按照数据依赖关系划分为多个程序块,将不同的程序块分别映射到不同的处理单元上完成局部计算,计算完成后按照数据依赖关系将计算结果传递给目标程序块,目标程序块利用该结果进行进一步计算. 在粗粒度数据流结构中,每个处理单元内的指令执行模式通常仍然为冯诺依曼执行模式. 细粒度数据流结构有TRIPS[8]、WaveScalar[13]等,细粒度数据流结构中每个处理单元的执行模式是数据流计算模式,程序直接被转换为数据流程序,指令之间通过数据建立依赖关系,指令执行的结果直接传递到目的指令. 数据流程序被映射到执行阵列中,每个阵列上分配数据流程序中的若干指令,这些指令可以是相关的也可以是无关的. 由于本文研究的数据流结构属于细粒度数据流结构,且其结构与TRIPS[8]结构相似,因此本文以TRIPS为例,介绍传统数据流处理器的结构和TRIPS上的指令映射算法.

如图1所示,TRIPS处理器是由4×4个执行单元通过二维MESH网络连接组成,相邻网络节点之间传递数据会出现一定的延迟. 缓存和寄存器堆被放置在执行单元阵列的周围,通过MESH网络与执行单元阵列相连接. 每个执行单元包含了指令缓存、操作数缓存、定点部件、浮点部件和路由逻辑. TRIPS支持8个程序块(Frame)的同时执行,每个程序块最多为128条指令. 指令缓存和操作数缓存被划分为多份,分别对应了不同的程序块. 程序块的指令被映射到4×4执行单元阵列上,每个执行单元执行被分配到的部分指令,并将执行结果传递给目的指令. 程序块执行完成后,TRIPS调度下一个程序块,将下一个程序块的指令映射到执行阵列上执行. 在数据流程序中,指令的源操作数存储空间都是私有的,因此在指令中不需要对源操作数地址进行编码,通常仅需要对源操作数的个数进行编码[14-15]. 另外,在数据流计算模式中,指令执行的结果直接送往消耗此结果的指令. 因此,在指令中需要对目的指令进行编码. 一般情况下,指令是有多个目的指令的.

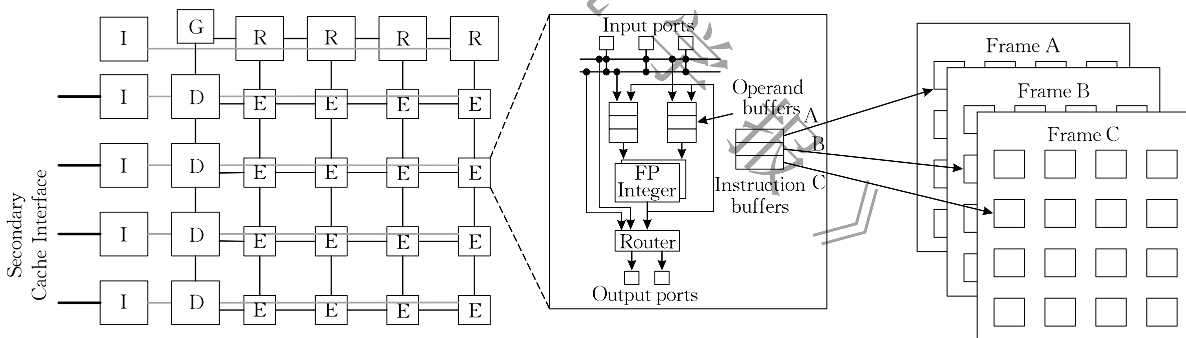

图1 TRIPS处理器架构[8]

在TRIPS中,每个程序块最多支持128条指令,这128条指令分别对应128个物理位置,因此指令编码中需要7位来表示一条指令的位置,指令的二进制编码必须在确定所有指令映射的位置之后才能确定下来. 图2显示了一个简单的数据流程序映射到2×2执行单元阵列上的位置,该程序为$c=(a+b)\times(a-b)$,其对应的数据流图如图3所示,其中a,b和c均为内存中的变量. 每个执行单元上最多可以放置2条指令,则程序所对应的指令编码如图2右侧所示,其中每条指令后面的四元组表示指令执行的结果需要传递到哪条指令的哪条操作数,其中四元组的第1个和第2个元素分别表示目的指令所在执行单元的x坐标和y坐标,第3个元素表示目的指令是执行单元上的第几条指令(每个执行单元上有2条指令,包括没有分配指令的空位置),第4个元素表示数据应该传递到目的指令的第几个操作数. 例如,LD0指令被映射到执行单元(0,0)的第0条指令位置,LD0指令执行完成后,其结果将会被传递到执行单元(0,1)上的第0条指令的第0个操作数和执行单元(0,1)上的第1条指令的第0个操作数. 在数据流执行模式中,所有指令的执行都是由数据驱动的,数据流指令在源操作数全部准备好之后即可执行,执行的结果被直接送往目的指令.

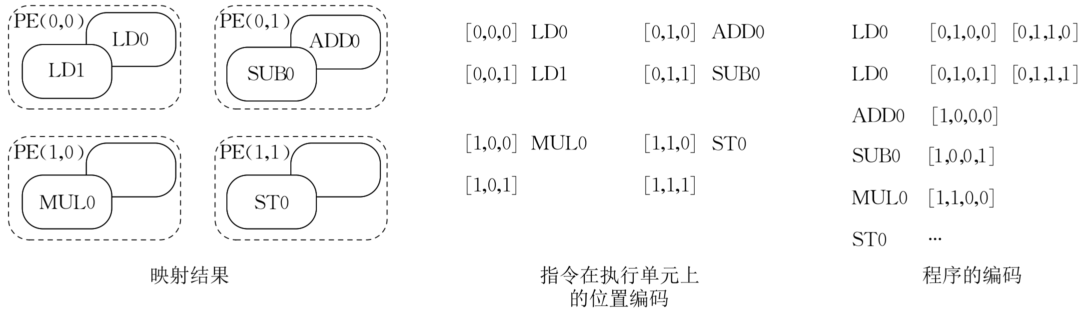

图 2 数据流图的程序编码和映射结果

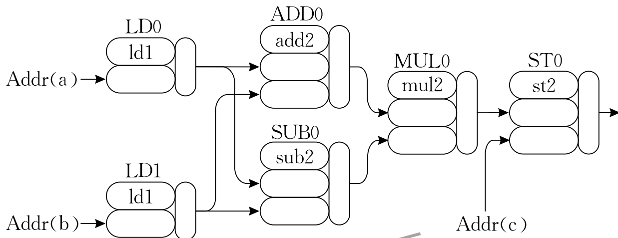

图 3 $c=(a+b)\times(a-b)$的数据流图

2.2 基于传统数据流结构的指令映射算法

基于 TRIPS[8] 结构，SPDI 指令映射算法[6] 和 SPS 指令映射算法[7] 相继被提出. 基准 SPDI 算法按照数据流图中节点的高度依次映射所有的指令，对每个节点计算其最佳位置. SPDI 对每一个可放置指令的执行单元计算其可执行时间，然后选出可执行时间最短的执行单元作为最佳映射位置. 可执行时间依赖于输入指令(数据流图中的父节点)的可执行时间、输入指令的执行延迟和数据的路径延迟. 另外，SPDI 还考虑了关键路径、功能单元竞争和访存指令的优先位置等因素，对可执行时间进行了适度的调整，使 SPDI 映射算法具有更高的性能.

SPS 算法在选择指令时考虑了更多可能的指令. SPS 将所有没有输入的指令和至少有一个输入且已经被调度的指令作为备选集合. 并且，在计算指令代价的时候考虑了备选集合中所有的指令，对于每条指令综合考虑了指令的输入延迟、执行延迟和输出延迟. SPS 对备选集合中所有的指令进行遍历，分别计算每条指令的最佳映射位置以及其指令代价. 然后选出最佳映射位置代价最大的指令作为当前调度对象，将其映射到最佳执行单元上. 在计算指令代价时，SPS 相比 SPDI 额外考虑了输出延迟，这体现了 SPS 的全局映射思想. SPS 不仅考虑了已经映射的指令，对未映射的指令位置也加以考虑，从全局角度来降低程序的执行时间.

3 循环流水优化方法

传统数据流结构中，一个上下文对应一个程序块，程序块与程序块之间互相切换执行. 只有当一个正在执行的程序块结束后，新的程序块才会被分配到执行阵列上准备执行. 在这种情况下，执行部件的利用率不高，在特定时间只有一部分指令在执行，而其他指令均在等待操作数的到来. 如图 4 左侧所示的数据流图，假设其在两个执行单元上执行，若将两个分支分别映射到不同执行单元上，两个执行单元并行执行不同的分支，则其计算部件利用率最高. 但是，只有在数据流图非常规则的情况下，传统数据流结构的执行部件利用率才会较高. 而通常情况下，数据流图并不规则. 将如图 4 右侧所示的数据流图映射到两个执行单元上，则不论怎么映射，其执行单元利用率都不高，因为在某个时间点，只有一个执行部件在执行指令，另外一个执行部件正在等待操作数的到来. 通过多线程多指令的方式虽然能稍微缓解执行单元利用率低的问题，但最终执行部件的利用率仍然不高.

《计算机学报》

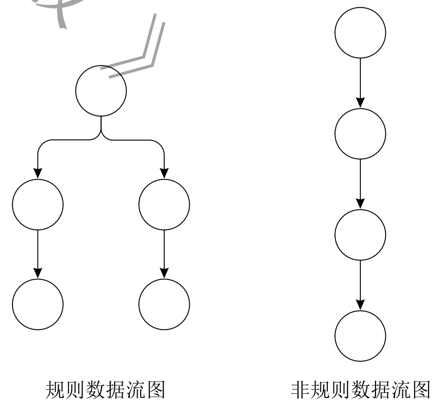

图 4 规则数据流图和非规则数据流图

在科学计算中，程序通常由核心(Kernel)构成，核心程序段是整个科学计算的主体部分，也是需要加速处理的部分. 这些核心程序通常简单但处理的数据量非常庞大. 它们通常由循环构成，对每一个点或者每一小块数据都进行同样的操作. 例如 2d5p Stencil 中，程序对矩阵中每一个点都进行 2 次乘法和 4 次加法运算. 在科学计算中，数据可以通过分块

进行处理,而且分块后每次循环处理一个子块,通常子块之间是没有数据依赖的,因此分块后的子块与子块之间可以并行处理.

传统数据流结构在处理科学应用时,通过循环来处理多个数据子块,循环执行一次处理的数据对应一个子块,当一个子块处理完成后,下一个子块才开始处理.这种处理方式没有利用子块之间可以并行处理的特征,而传统数据流结构通常由于等待操作数导致执行单元空闲,因此循环流水的优化方法将科学计算中的循环采用硬件自动迭代实现,将科学计算中的一次循环转换为一个上下文.循环流水的优化方法进一步利用科学计算中循环的并行处理特征,修改数据流结构中上下文切换逻辑,使每一个上下文的开始不需要等待上一个上下文的结束,即科学计算中的一次循环开始不需要等待上一次循环的结束.这样循环流水化结构上的多个上下文以流水线的方式进入执行阵列,数据流图中的每一条指令在执行一次后会立刻接收到下一个上下文的操作数,使执行单元的利用率得到极大的提高.

在循环流水优化后的数据流结构中,一个数据子块对应一个上下文,不同上下文对应不同的数据子块(图5),但它们对应的程序均为同一个数据流图.如图6,类似于指令在流水线中的执行方式,多个上下文以流水线的方式流过整个数据图,结果以流的方式从数据流图中流出.在优化后的数据流结构中,一个新的上下文的产生不再依赖于一个正在执行的上下文的结束,而仅仅依赖于正在执行的上下文的第一条指令的执行完成.如图6,OP4执行完上下文1的数据操作后,立刻接收到上下文2的操作数,可以再次执行运算.用户只需要将上下文控制逻辑中需要计算的上下文数量配置完成,启动计算后,上下文控制逻辑将上下文流水送入执行阵列,并统计已经完成的上下文数量,确定当前计算是否完成.

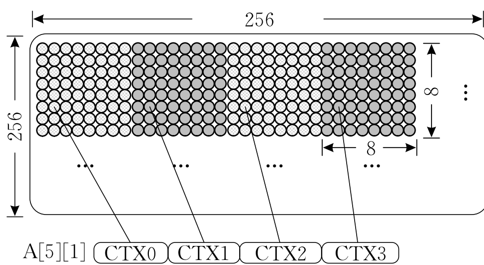

图5 循环流水优化后的上下文

在传统数据流中,由于只有少量上下文同时在执行,且大多数指令均在等待操作数,因此传统数据流结构的网络中传递的消息量不大.但是在循环流

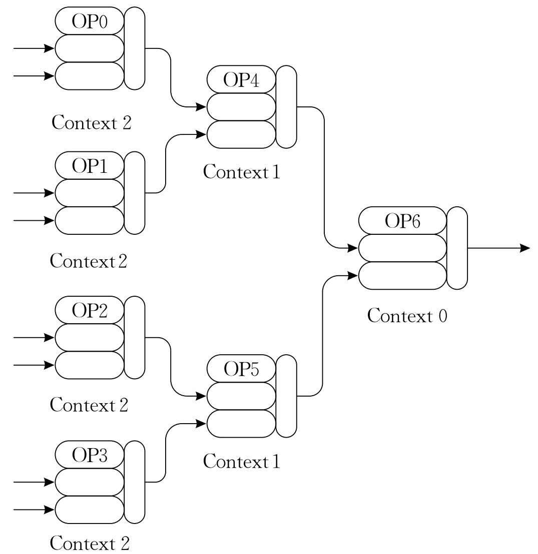

图6 循环流水化后上下文的执行方式

水优化后的数据流结构中,同时有成百上千的上下文在执行,这导致传递操作数的网络上的消息量暴增.因此,循环流水优化后的数据流结构对网络传输带宽要求很高,网络传递消息的效率对结构的性能有着至关重要的作用.如果由于网络拥堵导致操作数没有及时达到目的指令,这将导致执行单元上没有指令可以执行,从而降低了数据流结构的效率.因此,在循环流水优化后的数据流结构中,应该通过增加网络带宽来提高数据流结构的性能.

循环流水优化方法对分块和并行处理的科学应用是适用的,对于其他类型的应用无法进行优化处理.常见的科学计算,例如Stencil、FFT和矩阵乘法等都是可以通过分块和并行进行加速处理的.

4 LBC指令映射算法

在循环流水优化后的数据流结构中,传统的指令映射算法的性能不好,主要是由于优化后的数据流结构中流水线执行模式的加入.流水线执行模式的加入,使数据流图不再只执行一遍,而是以流水线的方式执行很多遍.因此,数据流图执行一遍的时间不再是性能的唯一标准.在传统数据流结构上,通过计算两条指令的可执行时间即可推测出指令之间的竞争.而循环流水优化后的数据流结构中,每条指令都会执行很多次,资源竞争非常激烈.即使可以推算出指令的第一次执行时间,但该指令的后续执行时间难以推算.

在循环流水化的数据流结构中,每条指令几乎在每一拍都具备执行的条件,指令对执行单元的竞

《计算机学报》

争很大.另外，每个执行单元几乎每一拍都会产生一个结果，执行单元之间相互传递的数据量非常大，导致片上网络的竞争也非常激烈.从提高执行单元和网络的有效利用率的角度出发，本文提出了面向循环流水化数据流结构的 LBC 指令映射算法. LBC 以执行单元负载为核心，同时考虑定点浮点和网络拥堵等影响因素，将所有指令按照一定规则映射到执行单元阵列上，提高了执行单元和网络的有效利用率，从而提高了循环流水优化后的数据流结构的性能.

4.1　执行单元负载均衡

在循环流水化的数据流结构中，每个执行单元的利用率很高.因此想要提高整个执行部件的利用率，负载均衡是最重要的.负载如果不均衡，将会导致执行单元完成计算任务的时间不一致，负载较低的执行单元则会出现空闲.

如图 7，将数据流图映射到 2 个执行单元上.传统指令映射算法映射的结果如图 7 中的(2)所示，其优先考虑关键路径(1,2 和 4)，要求指令之间的通信延迟短(1,2 和 4)，且同一个执行单元上的指令之间不会竞争(2 和 3 竞争)，这样数据流图执行一遍需要的时间较短.执行单元负载为核心的指令映射算法要求负载均衡，因此其映射的结果如图 7 中的(3)所示，每个执行单元上均为两条指令.在循环流水化的数据流结构中，数据流图会执行多遍，在传统指令映射方式下，执行单元 0 的负载是执行单元 1 的负载的三倍，因此，完成计算任务时，执行单元 0 的计算量是执行单元 1 的三倍.在传统数据流结构上，由于数据流图只执行一遍，因此三倍计算量带来的影响只有若干拍.在考虑网络延迟的同时，这三倍计算量带来的影响几乎与网络延迟相近，综合考虑网络延迟和执行单元负载，在传统数据流结构上这种映射方式是合理的.但是在循环流水化的数据流结构上，这三倍计算量带来的性能影响远远超过了其他因素带来的影响.

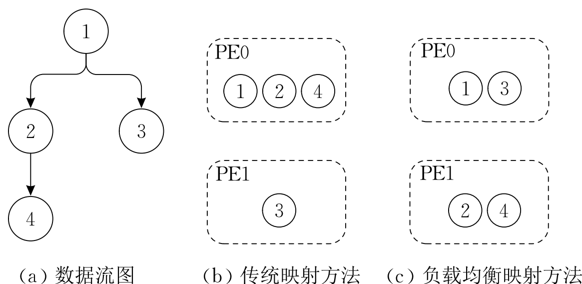

图 7　负载均衡映射方法和传统映射方法

如图 7，假设整个计算任务会执行该数据流图 N 次，且执行单元上仅有一个计算部件，则在循环流水化的数据流结构上，传统映射算法的执行时间约为 $N\times 3$ 拍.这是由于执行单元 1 执行完 $N\times 3$ 条指令需要 $N\times 3$ 拍，在循环流水化的数据流结构中，指令之间的通信与指令执行并行进行，在网络不出现拥堵的情况下，基本不会影响程序最终的执行完成时间.同理，负载均衡映射算法的执行时间约为 $N\times 2$ 拍.而通常 N 较大，因此在本例中，执行单元负载均衡的映射算法较传统映射算法能提高循环流水化数据流结构的性能约为 33.3%，执行单元负载均衡的指令映射算法在循环流水化数据流结构上具有较好的性能表现.

4.2　定点浮点均衡

在数据流结构中，执行单元每拍能执行的指令数量是由执行单元中定点部件和浮点部件的数量决定的.在映射指令时，若不区分定点浮点指令，则不同执行单元上定点部件或浮点部件可能会出现负载不均衡.当不同执行单元上分配到的特定类型指令比例与它们的执行部件比例相差较大时，这些执行单元上的执行部件则会出现负载不均衡.例如，一个执行单元 PE0 中有 2 个定点部件和 1 个浮点部件，而另一个执行单元 PE1 中有 1 个定点部件和 2 个浮点部件，那么在指令映射时应该保证 PE0 与 PE1 上的定点指令数量比例约为 2∶1，而浮点指令数量比例约为 1∶2.

因此，在循环流水优化后的数据流结构上，必须将定点和浮点指令分开处理，做到所有执行单元上浮点部件和定点部件都负载均衡.若执行单元上的部件数量不一致，还需要根据每个执行单元的处理能力确定定点或浮点指令数量.

将如图 8 所示的数据流图映射到两个执行单元上，每个执行单元上均有一个定点一个浮点部件.其中，I1～I6 均为定点指令（I2 和 I6 为访存指令），F1～F2 均为浮点指令.若按照传统指令映射算法，指令 I1、I2、F1 和 F2 会被映射到执行单元 0 上，而指令 I3，I4，I5，I6 会被映射到执行单元 1 上.这样能够保证执行单元 0 和执行单元 1 并行执行不同的路径，使数据流图执行一遍所需要的时间较短.但是，这样的映射结果导致执行单元 0 和执行单元 1 上的定点部件和浮点部件的负载都不是均衡的.在循环流水化数据流结构中，执行单元 0 的浮点负载很高，而执行单元 1 的浮点负载为 0.在执行计算任务时，所有的浮点指令均由执行单元 0 完成，完全浪费了执行单元 1 的浮点计算资源.而且，执行单元 1 上的定点指令数量是执行单元 0 上的两倍，这导

《计算机学报》

致执行单元1的定点负载是执行单元0的两倍.负载的不均衡导致两个执行单元的有效利用率大幅下降.

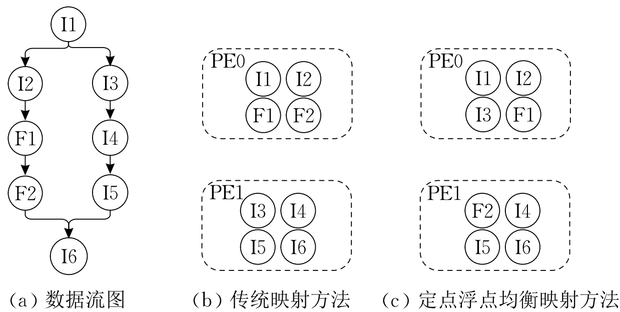

图8 定点浮点均衡映射方法和传统映射方法

假设数据流图执行 N 次,每个执行单元上有一个定点部件和一个浮点部件.在循环流水化数据流结构上,定点浮点均衡的映射算法的执行时间约为 $N\times 3$ 拍,传统映射算法的执行时间约为 $N\times 4$ 拍.因此,在循环流水化数据流结构上,定点和浮点指令应该分别处理,使得所有执行单元上定点部件和浮点部件的负载达到均衡,以降低计算任务的执行时间.尤其是在不同执行单元上定点部件和浮点部件不相等的情况下,指令映射算法需要按照执行单元的处理能力分配相应数量的指令,才能使程序的执行时间达到最短.

LBC指令映射算法通过分析负载代价来达到定点浮点负载均衡.负载代价的计算方式为 $h\times n$.其中 h 代表了数据流结构上相邻两个节点的数据传输延迟,即某个PE发送一条消息到相邻PE的传输延迟(不考虑拥堵),这个数值是一个结构确定的,在映射过程中是个常量.n 代表了PE上已经映射的特定类型指令(定点、浮点)的数量,假如当前需要映射一条浮点指令,则考虑某个PE时,n 等于此PE上已经映射的浮点指令数量.LBC算法采用 $h\times n$ 作为负载代价,在映射某条指令时,对所有的可用PE(PE上有可用的指令槽)分别计算其负载代价,选择负载代价最低的PE作为最佳映射位置,负载代价迫使选择指令位置时考虑负载更加均衡的映射位置.假设PE0上已经映射了8条指令,其相邻PE1上映射了7条指令,且待映射指令的父节点在PE0上,那么负载代价将会使待映射指令选择PE1,而不是PE0.而传统映射算法不考虑负载均衡,它将会优先选择PE0,因为PE0的数据传输时间较小.而根据前面的分析,选择PE1的程序执行性能会高于PE0.

4.3 网络竞争

在4.1节和4.2节的例子中,指令数量非常少,因此,网络传输的数据量也非常少,网络基本不会产生拥堵,故忽略掉了网络传递信息的延迟.在循环流水化数据流结构中,每个执行单元几乎每拍都会执行指令,因此网络上传递的数据量非常庞大.而且在通常的实际科学应用中,数据流图较大,分配到每个执行单元上的指令数量也较多,指令之间通信频繁,也加剧了网络上传递的数据量.提高网络信息传递的效率,避免网络产生拥堵,是提高循环流水化数据流结构性能的一个重要因素.在网络结构既定的基础上,如何降低数据传输量成为提高网络效率的关键.指令映射决定了执行单元之间传递信息的数量和距离,从而决定了网络上传输的总信息量,因此指令映射算法对网络传输的效率起着至关重要的作用.

在映射指令时,互相依赖的指令应该被映射到距离较近的或者同一个执行单元上,这样指令之间通过网络传递的信息较少或全无.网络上传递消息的数量由指令之间的位置决定的,距离较近的指令之间传递消息时,占用的网络资源较少,消息仅需要较少的几次传递即可到达目的地,单个消息所需要的网络资源较少;而距离较远的指令传递消息时,指令之间的多段网络路径会被占用,单个消息所需要的网络资源较多.因此,降低网络竞争的指令映射算法的基本原则为:互相依赖的指令应该尽量映射到距离较近的执行单元上.

但是,每个执行单元上所能容纳的指令数量是固定的,一旦到达容量上限,则该执行单元上不能映射新的指令.如果刻意将依赖的指令映射到相近或相同执行单元上,会导致一部分执行单元很快被填满,而其他执行单元上几乎没有指令.在这种情况下继续映射剩余没有映射的指令,则指令的位置受到了很大的限制,导致依赖的指令可能无法映射到较近的位置,产生了网络资源的浪费.因此,合理的指令映射算法应该全局综合考虑所有指令的映射位置,而不应该仅考虑局部最优的结果.在映射指令时,既要将依赖的指令映射到较近的位置,同时也要使执行单元上的指令在映射过程中始终处于均衡的状态,不会出现局部占满和局部空闲的状态.

在LBC指令映射算法中,传输代价表示为待映射节点的所有父节点到待映射节点的距离之和.传输代价采用父节点与子节点之间的传输延迟来衡量,是因为传输延迟代表着网络资源占用情况.因此父节点与子节点之间的传输延迟越小,则网络资源占用越少,网络竞争出现的概率越低,网络的传输性能越好.

《计算机学报》

4.4　LBC 指令映射算法

综合 4.1 节至 4.3 节中的优化方法，将 LBC 指令映射算法描述为算法 1. 由于数据在数据流图中从根节点流向叶节点，因此首先将数据流图中的指令按照深度（节点与根节点的距离）的顺序排序，优先映射深度较小的指令. 按照深度的顺序进行映射时，待映射节点的父节点必然已经完成映射，因此可以计算出待映射节点与父节点之间在 PE 阵列上的距离. LBC 算法首先按照深度对所有指令进行排序，按照深度顺序依次映射每条指令（第 1 层 FOR 循环）. 对于每条待映射指令，考虑执行单元阵列上的所有 PE（第 2 层 FOR 循环），分别计算待映射指令映射到此 PE 上的映射代价. 计算映射代价时，跳过无法放置更多指令的 PE. 最后，从所有 PE 中选出映射代价最小的 PE 作为最佳映射位置.

算法 1.　LBC 指令映射算法.

输入：*D*-数据流图，*A*-执行单元阵列

输出：*M*-*D* 到 *A* 之间的映射关系

1. *SD*=*SortByDepth*(*D*)
2. FOREACH instruction *i* in *SD* DO
3. 　*bestCost*=MAX；*bestPE*=none
4. 　FOREACH PE *p* in *A* DO
5. 　　IF *hasLegalSlot*(*p*) THEN
6. 　　　*Cost*(*i*,*p*)=*transferCost*(*i*,*p*)+*loadCost*(*i*,*p*)
7. 　　　IF *Cost*(*i*,*p*)<*bestCost* THEN
8. 　　　　*bestCost*=*Cost*(*i*,*p*)
9. 　　　　*bestPE*=*p*
10. 　　　END IF
11. 　　END IF
12. 　END FOR
13. 　*SD*=*SD*−{*i*}
14. 　*M*+=*map*(*i*,*bestPE*)
15. END FOR

映射代价的计算主要包含两个部分，*transferCost*（传输代价）和 *loadCost*（负载代价），其计算方式分别如 4.2 节和 4.3 节所示. 负载代价体现了负载均衡和定点浮点均衡的映射思想，传输代价体现了网络竞争优化的映射思想. 负载代价将定点和浮点分开处理，通过最小代价使每个部件在映射过程中达到负载均衡，使执行阵列在映射过程中不会出现局部满或局部空的状态.

如图 9 所示，将一个简单的数据流图（如图 9(a)）映射到两个 PE 上，每个 PE 上最多可以容纳 6 条指令，且每个 PE 上有一个定点部件和一个浮点部件. 根据传统指令映射算法 SPDI 和 SPS，其映射结果如图 9(b)和图 9(c). 其最终映射的结果中 PE 负载不均衡，且不同 PE 上定点部件和浮点部件的负载也不均衡，在循环流水优化后的数据流结构上，这将会产生性能损失.

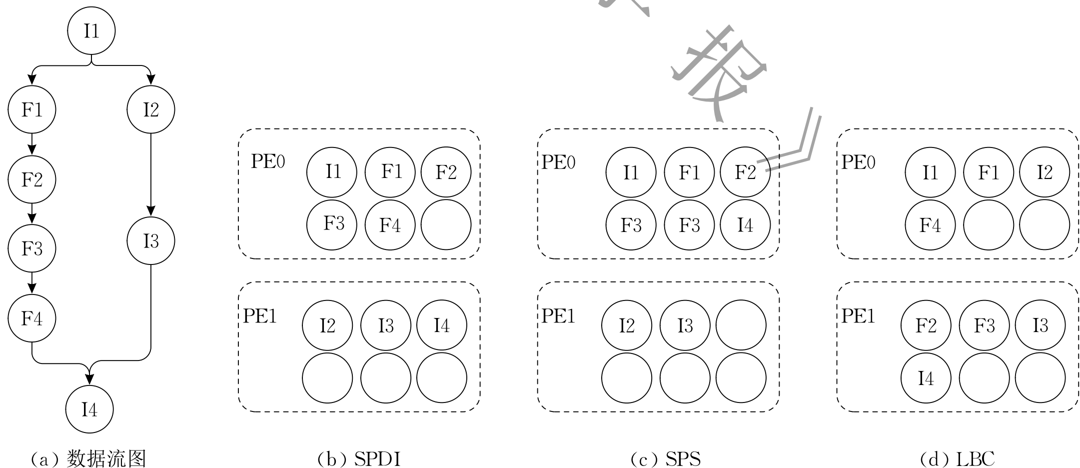

图 9　SPDI 算法、SPS 算法与 LBC 算法

传统指令映射算法只考虑数据的传输延迟和指令执行部件是否产生竞争，依次计算每条指令在不同 PE 上的执行时刻，确定指令的最佳映射位置. 在数据流图仅执行一次时，SPDI 和 SPS 算法的核心是非常合适的，每条指令的执行时刻都可以通过模型确定下来. 然而在循环流水优化后的数据流结构上，数据流图会执行多次，因此每条指令的执行时刻不能通过简单的模型计算出来. 而 LBC 指令映射算法从全局上考虑指令的执行方式，从而确定以负载均衡为核心的指令映射方法. LBC 指令映射算法的映射结果如图 9(c)所示，PE0 和 PE1 的负载相同，且不同 PE 上定点指令数量和浮点指令数量均相同. LBC 指令映射算法以负载均衡为核心，且同时考虑了不同 PE 上定点部件和浮点部件的负载均

计算机学报

衡，并进一步通过传输代价使网络通信量得到降低，从而提高数据流程序在循环流水优化后的数据流结构上的执行性能.

5　实验及分析

5.1　实验平台

实验平台以中国科学院计算技术研究所自主研发的大规模并行模拟框架 SimICT[16] 为平台，运行本文研究所研发的数据流处理器模拟器. 数据流处理器模拟器的结构如图 10 所示，主要包含 ARM 处理器、DMA 控制器、内存和数据流加速部件等模拟组件. 实验环境具体配置如表 1. 由于处理器对通用部件的要求很低，通用部件作为辅助功能部件为加速部件服务. 因此，模拟器中通用部件采用简单的 ARM 顺序多发射处理器. 另外，由于科学计算中通常需要对多维数据进行处理和传输，因此 DMA 控制器中加入了带步长的三维数据传输功能，带步长的三维数据传输为数据分块处理提供了有力的支持. 循环流水化数据流加速部件采用 8×8 的 PE 阵列来加速应用的处理能力，提高处理器的峰值性能. PE 之间通过 2D MESH 网络连接，由于循环流水优化后的数据流加速部件的数据注入率较高，网络路由采用4套 256 位 MESH 网络来完成 PE 之间的数据交互. 每套网络独立传输数据，不支持跨网络的数据传递. 加速部件上采用 8M 缓存，通过缓存控制器与 8 个路由相连.

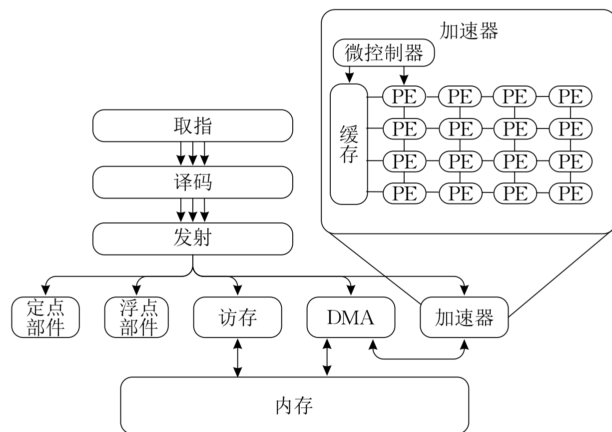

图 10　数据流处理器架构

表 1　实验环境配置

模拟器配置	
CPU	1 GHz，顺序多发射，ARM 指令集
DMA	支持三维带步长数据传输
内存	256 GB/s
PE	64 个，SIMD4，256 位，1 GHz
MESH 网络	4 套独立网络，每套网络 256 bit 数据总线，相邻节点间 1 拍延迟
缓存	128 Bank，8 MB，128×64b
双精度峰值	512 GFLOPS

另外，模拟器中各项操作完成所需要时间周期数如表 2.

表 2　各项操作时钟周期数

操作	周期数
浮点加法	1 拍
浮点乘法	3 拍
浮点乘加	4 拍
内存访问周期	100 拍
2D MESH	1 拍/跳

ARM 处理器将数据流指令和计算所需的数据通过 DMA 从内存拷贝到加速部件中，通过配置加速部件微控制器中的控制位来启动加速部件，计算完成后通过 DMA 将结果拷贝回内存. 数据流指令的映射是在汇编时完成的，加速部件根据映射结果将指令分配到各执行单元中. 本实验将指令映射算法加入数据流汇编程序中，在生成二进制可执行程序时同时将指令的位置信息写入二进制文件. 二进制文件通过 DMA 拷贝到加速部件，加速部件中的微控制器通过读取指令信息，确定指令的分配位置.

实验以 4 种常见的科学计算（双精度浮点）2D Stencil、3D Stencil、FFT 和矩阵乘法作为测试程序，测试循环流水优化方法的性能表现，以及不同指令映射算法在循环流水化数据流结构上的性能. 测试程序配置如表 3，其中分块大小表示数据流程序执行一遍处理的数据分块大小，TILE 规模为每次启动加速部件计算的数据规模. 计算任务首先以 TILE 规模进行 TILE 划分，多个 TILE 通过双缓冲机制实现数据 DMA 传输和加速部件计算的并行执行，从而不降低数据流加速器的性能. 对于每个 TILE 的数据，以分块大小为基本块进行分块处理，每块对应循环流水优化后的数据流结构中的一个上下文.

表 3　测试程序配置

双精度浮点测试程序	分块大小	TILE 规模
FFT	32 点 FFT	1024×32
2D Stencil	8×8 2d5p Stencil	128×128
3D Stencil	8×8×32 3d7p Stencil	64×64×32
矩阵乘法	8×128 与 128×8 矩阵乘法	128×128

实验首先将传统数据流结构和循环流水优化后的结构进行性能对比，采用的指令映射方式均为传统的 SPDI 和 SPS 指令映射方法，从而体现出循环

《计算机学报》

流水优化所带来的性能提升. 实验中模拟的传统数据流结构的执行方式与 TRIPS 执行方式相同，参数配置与循环流水化的数据流结构一致，均如表 1 所示. 两者的主要区别在于：在循环流水化结构中，多上下文以流水线的执行方式进行，而在 TRIPS 执行方式中，一个上下文的开始需要等待另一个正在执行的上下文的结束.

其次，实验将详细分析 LBC 指令映射算法对循环流水优化后的结构带来的性能提升，以及与传统 SPDI 和 SPS 指令映射算法的性能对比. 实验将 LBC 指令映射算法中的负载均衡(LB)、定点浮点均衡(FFB)和网络竞争(NC)逐步加入到指令映射算法中，分析 3 种优化分别对性能的影响.

另外，由于循环流水化数据流增大了网络负载，网络带宽极大地影响了循环流水化数据流结构的性能，因此本实验也将分析不同网络带宽的配置下，LBC 指令映射算法的性能变化.

在科学计算领域，计算效率是衡量计算机体系结构的重要指标，计算效率代表了体系结构对计算资源的利用情况. 在功耗逐渐成为高性能计算的主要问题时，计算效率显得尤为重要. 因此，本文从计算效率的角度出发，将本文提出的数据流结构优化方法优化后的数据流结构效率与 GPU 结构效率进行对比，从而体现出数据流结构在科学计算领域的优势. 由于科学计算中大多为浮点计算，因此本文所指的计算效率为浮点效率. 计算效率的计算方法为

$$\text{计算效率}=\frac{\text{测试程序性能}}{\text{结构的理论峰值性能}}\times 100\%,$$

其中，测试程序性能表示测试程序在结构上取得的浮点性能(GFLOPS)，结构的理论峰值性能表示硬件架构所能取得的理论峰值性能(GFLOPS).

5.2　实验结果及分析

表 4 显示了循环流水优化后的数据流结构与传统数据流结构在不同网络带宽下的性能表现(GFLOPS)，其中分别采用了两种传统的指令映射方式 SPDI 和 SPS. 在 4 类程序中，循环流水优化后的数据流结构相比传统结构均有一定的性能提升，其中 FFT 提升较大，平均达到 14.47%，4 种测试程序平均提升了 4.6%. 相比 SPDI，SPS 指令映射算法在循环流水优化后的数据流结构上的性能更好.

表 4　循环流水化结构和传统结构在不同网络带宽下的性能

		指令映射方法	FFT	2D Stencil	3D Stencil	MatrixMul
2MESH	传统数据流结构(GFLOPS)	SPDI	57.54	36.82	36.71	56.81
	循环流水化的数据流结构(GFLOPS)	SPDI	64.39	38.14	36.79	56.99
	性能提升/%		11.90	3.57	0.23	0.33
	传统数据流结构(GFLOPS)	SPS	57.74	56.68	43.72	57.90
	循环流水化的数据流结构(GFLOPS)	SPS	58.01	57.45	43.80	57.85
	性能提升/%		0.47	1.36	0.20	−0.09
4MESH	传统数据流结构(GFLOPS)	SPDI	63.58	46.61	52.67	111.83
	循环流水化的数据流结构(GFLOPS)	SPDI	77.65	47.10	52.91	112.80
	性能提升/%		22.12	1.07	0.45	0.87
	传统数据流结构(GFLOPS)	SPS	76.75	85.31	62.05	90.81
	循环流水化的数据流结构(GFLOPS)	SPS	83.56	87.61	62.83	91.22
	性能提升/%		8.87	2.70	1.26	0.45
6MESH	传统数据流结构(GFLOPS)	SPDI	66.95	60.30	61.47	124.54
	循环流水化的数据流结构(GFLOPS)	SPDI	83.18	61.67	62.55	125.72
	性能提升/%		24.25	2.28	1.75	0.95
	传统数据流结构(GFLOPS)	SPS	77.44	87.80	62.12	124.97
	循环流水化的数据流结构(GFLOPS)	SPS	84.37	89.43	62.85	125.93
	性能提升/%		8.94	1.85	1.18	0.77
8MESH	传统数据流结构(GFLOPS)	SPDI	66.69	59.59	61.44	124.54
	循环流水化的数据流结构(GFLOPS)	SPDI	83.08	61.46	62.53	125.78
	性能提升/%		24.58	3.14	1.78	1.00
	传统数据流结构(GFLOPS)	SPS	75.35	88.17	62.16	124.94
	循环流水化的数据流结构(GFLOPS)	SPS	86.34	90.34	62.85	125.86
	性能提升/%		14.59	2.47	1.12	0.74
平均提升/%			14.47	2.30	1.00	0.63

但是，从整体来看，循环流水优化带来的性能提升较小，且网络带宽的增加也并没有明显提升结构的性能. 经过分析发现，这是由于传统指令映射算法不适合循环流水优化后的数据流结构，导致循环流水优化的架构优势没有体现出来.

调整指令映射算法，采用本文提出的 LBC 指令

计算机学报

映射算法后，循环流水优化后的数据流架构的性能得到了充分的发挥. 图 11 是各种指令映射算法在 MESH 网络为 8 套网络的配置下通过模拟测试得出的循环流水化数据流结构的性能. 对于每一种测试程序，其中 SPDI 和 SPS 分别表示传统的 SPDI 算法和 SPS 算法，LB 表示单独使用负载均衡优化时的性能，NC 表示单独使用网络竞争优化时的性能，LB＋FFB 表示负载均衡优化和定点浮点均衡优化同时使用时的性能，LB＋NC 表示负载均衡优化和网络竞争优化同时使用时的性能，LB＋FFB＋NC 表示负载均衡优化、定点浮点均衡优化和网络竞争优化都采用时取得的性能，即本文提出的 LBC 指令映射算法 LBC＝LB＋FFB＋NC.

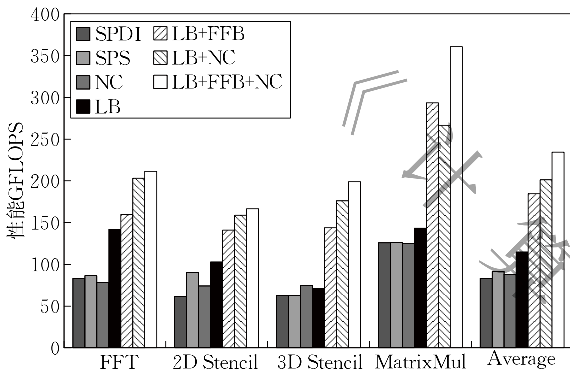

图 11　各指令映射算法的性能

对于 FFT、2D Stencil、3D Stencil 和矩阵乘法，单独使用 LBC 算法中的网络竞争优化取得的性能分别为 78.3GFLOPS、74.0GFLOPS、75.0GFLOPS 和 124.6GFLOPS，而加入了负载均衡优化后，4 种测试程序的性能分别提升了 159.6%、114.5%、134.8% 和 114.0%，负载均衡所带来的性能提升平均为 130.7%，由此可见负载均衡对循环优化后的数据流结构的性能提升非常显著.

单独使用 LBC 算法中的负载均衡取得的性能分别为 141.8GFLOPS、102.9GFLOPS、71.3GFLOPS 和 143.4GFLOPS，而加入了定点浮点均衡优化后，4 种测试程序的性能分别提升了 12.5%、37.0%、101.7% 和 104.6%，定点浮点均衡优化带来的性能提升平均为 63.9%. 最后在负载均衡和定点浮点均衡的基础上加入了网络竞争优化后，4 种测试程序的性能进一步提高了 32.6%、18.0%、38.4% 和 22.9%，分别达到了 211.6GFLOPS、166.4GFLOPS、198.9GFLOPS 和 360.6GFLOPS，网络竞争优化带来的性能提升平均为 28.0%. 在采用 LBC 指令映射算法后，4 种测试程序的性能分别比 SPDI 提高了 154.7%、170.8%、218.1% 和 186.7%，平均比 SPDI 算法提高了 182.6%，4 种测试程序的性能分别比 SPS 提高了 145.0%、84.2%、216.5% 和 186.5%，平均比 SPS 提高了 158.1%.

表 5 显示了在不同网络配置下，不同指令映射方法的性能表现. 表中 SPDI、SPS、NC、LB、LB＋FFB、LB＋NC 和 LB＋FFB＋NC 与图 11 中表示的映射方法相同，后面的((LB＋NC)－NC)/NC 表示在网络竞争优化的基础上加入负载均衡优化后的性能提升(百分比)，((LB＋FFB)－LB)/LB 表示在负载均衡的基础上加入定点浮点均衡优化后的性能提升(百分比)，((LB＋FFB＋NC)－(LB＋FFB))/(LB＋FFB) 表示在负载均衡和定点浮点均衡的基础上加入网络竞争优化后的性能提升(百分比)，(LBC－SPDI)/SPDI 表示 LBC 指令映射算法相比 SPDI 的性能提升(百分比)，(LBC－SPS)/SPS 表示 LBC 指令映射算法相比 SPS 的性能提升(百分比).

在循环流水化数据流结构上，随着网络带宽的增加，测试程序的性能逐步提高，当网络增加到一定程度时，程序的性能不再有明显的提高，网络带宽不再是性能的瓶颈. 例如 3D Stencil 采用 LBC 指令映射算法时，在 6 套 MESH 网络和 8 套 MESH 网络下，它的性能分别为 194.9GFLOPS 和 198.9GFLOPS，两

表 5　不同网络配置下不同指令映射算法的性能

		FFT	2D Stencil	3D Stencil	MatrixMul	Average
2MESH	SPDI(GFLOPS)	64.39	38.14	36.79	56.99	
	SPS(GFLOPS)	58.01	57.45	43.80	57.85	
	NC(GFLOPS)	54.88	62.15	53.37	84.21	
	LB(GFLOPS)	58.49	55.84	31.01	47.33	
	LB＋NC(GFLOPS)	72.61	61.89	98.49	148.63	
	LB＋FFB(GFLOPS)	68.29	62.60	37.78	76.02	
	LB＋FFB＋NC(GFLOPS)	72.24	64.89	89.74	150.32	
	(((LB＋NC)－NC)/NC)/%	32.32	－0.42	84.52	76.50	48.23
	(((LB＋FFB)－LB)/LB)/%	16.76	12.11	21.81	60.63	27.83
	(((LB＋FFB＋NC)－(LB＋FFB))/(LB＋FFB))/%	5.78	3.65	137.54	97.74	61.18
	((LBC－SPDI)/SPDI)/%	12.19	70.15	143.89	163.75	97.49
	((LBC－SPS)/SPS)/%	24.53	12.95	104.86	159.86	75.55

(续　表)

		FFT	2D Stencil	3D Stencil	MatrixMul	Average
4MESH	SPDI(GFLOPS)	77.65	47.10	52.91	112.80	
	SPS(GFLOPS)	83.56	87.61	62.83	91.22	
	NC(GFLOPS)	75.54	73.25	75.00	124.69	
	LB(GFLOPS)	104.36	94.81	61.06	94.11	
	LB+NC(GFLOPS)	138.15	118.69	172.58	266.43	
	LB+FFB(GFLOPS)	124.76	122.84	74.15	150.27	
	LB+FFB+NC(GFLOPS)	135.99	124.18	171.98	295.97	
	(((LB+NC)−NC)/NC)/%	82.87	62.04	130.10	113.68	97.17
	(((LB+FFB)−LB)/LB)/%	19.56	29.57	21.44	59.67	32.56
	(((LB+FFB+NC)−(LB+FFB))/(LB+FFB))/%	9.00	1.09	131.93	96.96	59.75
	((LBC−SPDI)/SPDI)/%	75.13	163.63	225.03	162.38	156.54
	((LBC−SPS)/SPS)/%	62.75	41.75	173.71	224.46	125.67
6MESH	SPDI(GFLOPS)	83.18	61.67	62.55	125.72	
	SPS(GFLOPS)	84.37	89.43	62.85	125.93	
	NC(GFLOPS)	78.60	69.84	74.94	124.60	
	LB(GFLOPS)	139.23	101.80	71.26	139.50	
	LB+NC(GFLOPS)	192.84	157.54	175.58	266.79	
	LB+FFB(GFLOPS)	158.27	138.90	109.34	224.03	
	LB+FFB+NC(GFLOPS)	188.41	165.77	194.87	360.63	
	(((LB+NC)−NC)/NC)/%	145.34	125.56	134.29	114.12	129.83
	((((LB+FFB)−LB)/LB))/%	13.68	36.44	53.44	60.59	41.04
	(((LB+FFB+NC)−(LB+FFB))/(LB+FFB))/%	19.04	19.35	78.22	60.98	44.40
	((LBC−SPDI)/SPDI)/%	126.49	168.80	211.54	186.85	173.42
	((LBC−SPS)/SPS)/%	123.32	85.37	210.05	186.38	151.28
8MESH	SPDI(GFLOPS)	83.08	61.46	62.53	125.78	
	SPS(GFLOPS)	86.34	90.34	62.85	125.86	
	NC(GFLOPS)	78.26	74.04	74.96	124.64	
	LB(GFLOPS)	141.83	102.94	71.28	143.36	
	LB+NC(GFLOPS)	203.17	158.81	175.97	266.69	
	LB+FFB(GFLOPS)	159.56	140.99	143.76	293.33	
	LB+FFB+NC(GFLOPS)	211.57	166.41	198.90	360.63	
	(((LB+NC)−NC)/NC)/%	159.62	114.50	134.75	113.97	130.71
	(((LB+FFB)−LB)/LB)/%	12.50	36.97	101.69	104.61	63.94
	(((LB+FFB+NC)−(LB+FFB))/(LB+FFB))/%	32.59	18.03	38.35	22.94	27.98
	((LBC−SPDI)/SPDI)/%	154.65	170.76	218.06	186.72	182.55
	((LBC−SPS)/SPS)/%	145.04	84.19	216.45	186.53	158.05

《计算机学报》

种情况下的性能几乎相同.随着网络带宽的增加，LBC指令映射算法相比SPDI和SPS算法的性能提高也逐步增加并趋于稳定.由此可见，网络带宽给LBC指令映射算法带来的性能提升要比SPDI和SPS大.从LBC指令映射算法内部分析，我们可以看到，随着网络带宽的增加，负载均衡所带来的性能提升平均从48.2%逐步增加到130.7%，定点浮点均衡所带来的性能提升平均从27.8%逐渐提高到63.9%，而网络竞争优化所带来的性能提升平均从61.2%逐渐降低到28.0%.这是由于网络带宽的增加弱化了网络竞争优化所带来的效益，在较高的网络带宽下，网络竞争较小，网络竞争优化所带来的性能提升也就越小.而在网络带宽较高的情况下，负载均衡和定点浮点均衡成为主要的性能瓶颈，因此其带来的性能提升也就越大.

经过本文提出的循环流水优化和LBC指令映射优化后，数据流结构在处理典型的科学应用时能取得较高的效率.在处理FFT、2D Stencil、3D Stencil和矩阵乘法时，优化后的数据流结构取得的浮点性能分别为211.6GFLOPS、166.4GFLOPS、198.9GFLOPS和360.6GFLOPS，实验中模拟的数据流结构的理论峰值性能为512GFLOPS(双精度)，因此4种测试程序对应的效率分别为41.3%、32.5%、38.8%和70.4%.

对于FFT，文献[17]显示，在Tesla C2075(1.03TFLOPS)上，经过优化后的FFT的浮点性能为200GFLOPS，因此其效率为19.4%，经过优化后的数据流结构的效率为GPU效率的210%.对于2D Stencil，文献[18]显示，在Tesla C2050(1.03TFLOPS)上，经过优化后的2D Stencil的浮点性能为100GFLOPS，因此其效率为9.7%，经过优化后的数据流结构的效率为GPU效率的340%.

对于 3D Stencil,文献[19]显示,在 Tesla C2050 上,经过优化后的 3D Stencil 的浮点性能为 87.3GFLOPS,因此其效率为 8.5%,而较新的文献[20]显示,在 Tesla K20C(3.52TFLOPS)上,3D Stencil 的浮点性能为 175GFLOPS,其效率为 5.0%,这主要是因为较新的 K20C 架构在理论峰值性能提升较大,但其显存带宽提升较小.对于 Stencil 这种计算访存比较低的程序,其性能主要受显存带宽的限制,因此其性能并不是随着理论峰值性能线性增长,从而导致了 3D Stencil 在 K20C 上的效率更低.经过优化后的数据流结构的效率为 GPU 效率的 460%.对于矩阵乘法,文献[21]显示,在 Tesla S2050(1.00TFLOPS)上,经过优化后的矩阵乘法的浮点性能为 804GFLOPS,其效率为 78.3%,经过优化后的数据流结构的效率是 GPU 效率的 90%.

因此,在处理 FFT、2D Stencil、3D Stencil 和矩阵乘法时,经过优化后的数据流结构的效率分别为 GPU 效率的 210%、340%、460%和 90%,仅在矩阵乘法上略低于 GPU 的效率,而在其他常见程序上明显优于 GPU 的效率.这是由于 GPU 等传统高性体系结构以矩阵乘法为高性能基准测试程序,对矩阵乘法做了较多的优化,所以能取得较高的效率,而在其他未进行特殊结构优化的程序上,其效率则相对较低.

6 结论及未来工作

面向科学计算领域时,传统数据流结构没有利用科学计算的分块和并行处理特征,对不同上下文的串行处理导致了传统数据流结构的计算部件利用率较低.本文通过分析科学计算的特征,提出了一种面向科学计算的数据流结构优化方法:循环流水优化方法.循环流水优化方法通过将科学计算中的循环采用硬件自迭代的方式实现和流水化处理,使数据流结构中的计算部件利用率得到提升.实验结果表明,在传统指令映射方式下,相比于传统数据流结构,循环流水优化后的数据流结构的性能平均提升了 4.6%.

由于传统数据流指令映射方式不适合循环流水优化后的数据流结构,没有发挥出循环流水化的数据流结构的优点.本文针对循环流水优化后的数据流结构特征,进一步提出了一种面向循环流水化数据流结构的指令映射算法 LBC.LBC 将负载均衡作为循环流水化数据流结构上指令映射的核心标准,要求所有执行单元上的定点部件和浮点部件的负载达到均衡.同时,循环流水化数据流结构大幅增加了网络上传递的数据量,网络竞争优化提高了网络传递数据的效率.实验结果表明,在处理 Stencil、FFT 和矩阵乘法等科学应用时,相比于传统指令映射算法 SPDI 和 SPS,LBC 指令映射算法在循环流水化结构上将测试程序的性能分别平均提升了 182.6% 和 158.1%.

我们将进一步优化数据流结构,在保证性能不下降的前提下,降低网络代价.在指令映射方面,我们将进一步探索可优化的空间例如访存指令优化和循环指令优化等,使指令映射算法达到最佳.

致 谢 感谢中科院计算所国重实验室编译组对数据流处理器编译上的支持,感谢吴萌和李戈在数据流编程实现上的支持!

参 考 文 献

[1] Oriato D, Tilbury S, Marrocu M, et al. Acceleration of a meteorological limited area model with dataflow engines//Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing. Chicago, USA, 2012: 129-132

[2] Pratas F, Oriato D, Pell O, et al. Accelerating the computation of induced dipoles for molecular mechanics with dataflow engines//Proceedings of the 2013 IEEE 21st Annual International Field-Programmable Custom Computing Machines. Seattle, USA, 2013: 177-180

[3] Fu Haohuan, Gan Lin, Clapp R G, et al. Scaling reverse time migration performance through reconfigurable dataflow engines. IEEE Micro, 2013, 34(1): 30-40

[4] Theobald K B. Earth: An Efficient Architecture for Running Threads[Ph. D. dissertation]. McGill University, Quebec, Canada, 1999

[5] Milutinovic V, Salom J, Trifunovic N, Giorgi R. Guide to Dataflow Supercomputing. Switzerland: Springer, 2015

[6] Nagarajan R, Kushwaha S K, Burger D, et al. Static placement, dynamic issue scheduling for EDGE architectures//Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques. Antibes Juan-les-Pins, France, 2004: 74-84

[7] Coons K E, Xia Chen, Burger D, McKinley K S, et al. A spatial path scheduling algorithm for EDGE architectures//Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operation Systems. San Jose, USA, 2006: 129-140

[8] Burger D, Keckler S W, McKinley K S, et al. Scaling to the end of silicon with EDGE architectures. IEEE Computer, 2004, 37(7): 44-55

《计算机学报》

[9] Nguyen A, Satish N, Chhugani J, et al. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs//Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis. New Orleans, USA, 2010: 1-13

[10] Gu Liang, Li Xiaoming, Siegel J. An empirically tuned 2D and 3D FFT library on CUDA GPU//Proceedings of the 24th ACM International Conference on Supercomputing. Tsukuba, Japan, 2010: 305-314

[11] Giorgi R, Badia R M, Bodin F, et al. TERAFLUX: Harnessing dataflow in next generation teradevices. Journal of Microprocessors and Microsystems: Embedded Hardware Design, 2014, 38(8): 976-990

[12] Carter N P, Agrawal A, Borkar S, et al. Runnemede: An architecture for ubiquitous high-performance computing//Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture. Shenzhen, China, 2013: 198-209

[13] Swanson S, Schwerin A, Mercaldi M, et al. The WaveScalar architecture. ACM Transactions on Computer Systems, 2007, 25(2): 1-54

[14] Dennis J B. Data flow supercomputers. IEEE Computer, 1980, 13(11): 48-56

[15] Dennis J B, Gao G R. An efficient pipelined dataflow processor architecture//Proceedings of the ACM/IEEE Conference on Supercomputing. Orlando, USA, 1988: 368-373

[16] Ye Xiaochun, Fan Dongrui, Sun Ninghui, et al. SimICT: A fast and flexible framework for performance and power evaluation of large-scale architecture//Proceedings of the 2013 IEEE International Symposium on Low Power Electronics and Design. Beijing, China, 2013: 273-278

[17] del Mundo C, Feng Wu-Chun. Towards a performance-portable FFT library for heterogeneous computing//Proceedings of the 11th ACM Conference on Computing Frontiers. Cagliari, Italy, 2014: 1-10

[18] Holewinski J, Pouchet Louis-Noel, Sadayappan P. High-performance code generation for stencil computations on GPU architectures//Proceedings of the 26th International Conference on Supercomputing. Venice, Italy, 2012: 311-320

[19] Zhang Yongpeng, Mueller F. Autogeneration and autotuning of 3D stencil codes on homogeneous and heterogeneous GPU clusters. IEEE Transactions on Parallel and Distributed Systems, 2013, 24(3): 417-427

[20] Luo Yulong, Tan Guangming, Mo Zeyao, Sun Ninghui. FAST: A fast stencil autotuning framework based on an optimal-solution space model//Proceedings of the 29th International Conference on Supercomputing. Newport Beach, USA, 2015: 187-196

[21] Kurzak J, Tomov S, Dongarra J. Autotuning GEMM kernels for the fermi GPU. IEEE Transactions on Parallel and Distributed Systems, 2012, 23(11): 2045-2057

《计算机学报》

SHEN Xiao-Wei, born in 1987, Ph.D. candidate. His main research interests include processor micro-architecture and high-performance computer systems.

YE Xiao-Chun, born in 1981, Ph.D., associate professor. His main research interests include high-performance computer architecture and software simulation.

WANG Da, born in 1981, Ph.D., associate professor. Her main research interests include processor micro-architecture and VLSI design.

ZHANG Hao, born in 1980, Ph.D., associate professor. His main research interests include processor micro-architecture and high-performance computer systems.

WANG Fei, born in 1981, Ph.D., associate professor. His main research interests include processor micro-architecture and FPGA design.

TAN Xu, born in 1991, Ph.D. candidate. His main research interests include high-performance computer architecture and software simulation.

ZHANG Zhi-Min, born in 1963, Ph.D., professor, Ph.D. supervisor. His main research interests include SoC design and computer architecture.

FAN Dong-Rui, born in 1979, Ph.D., professor, Ph.D. supervisor. His main research interests include low-power design and processor micro-architecture.

TANG Zhi-Min, born in 1966, Ph.D., professor, Ph.D. supervisor. His main research interests include high performance computer architecture design and digital signal processing.

SUN Ning-Hui, born in 1968, Ph.D., professor, Ph.D. supervisor. His main research interests focus on high performance computer systems.

Background

Traditional dataflow architectures hide the latency of waiting for operands through multiple contexts. But multiple contexts can only improve the utilization of function units in processing elements of dataflow architectures partly. When

dealing with typical scientific applications such as stencil, FFT and matrix multiplication, they are still not efficient enough because the utilization of function units is not very high. In this paper, we propose an optimization of dataflow architectures for scientific applications. The optimization takes advantages of the blocking and parallelism features of scientific applications and improves the context control logic of traditional dataflow architectures. The optimization implements the loop of scientific applications in hardware and switches the contexts in pipeline mode. The loop-in-pipeline dataflow architecture streams the contexts into the processing element array in pipeline model. Experimental results show that the loop-in-pipeline optimization achieves a 4.6% average performance improvement over the traditional dataflow architectures.

But the traditional dataflow instruction mapping algorithms are not adapted to the loop-in-hardware dataflow architectures. Based on the features of the loop-in-hardware architectures, we propose a novel instruction mapping algorithm: LBC (load-balance-centric) dataflow instruction mapping algorithm. The LBC algorithm maps the instructions into the PE array according to the depth of each instruction in the dataflow graph. For each instruction, the LBC algorithm computes the cost of each position of the PE array and takes the position of the minimum cost as the best position. The LBC algorithm takes the multiplication of the transfer delay of adjacent PEs and the number of the same type (fixed of float) instructions which has been mapped to the PE as the load-cost. Besides, the LBC algorithm augments the algorithm with the consideration of network contention. The sum of the distances between an instruction and its predecessor instructions is taken as the transfer-cost. The transfer-cost makes the path of each message transferred in the on-chip network minimum and reduce the hops of messages in the network. Experimental results show that the LBC algorithm achieves a 157% and a 126% average performance improvement over SPDI and SPS algorithm respectively.

This project is supported by the National High Technology Research and Development Program (863 Program) of China (No. 2015AA01A301), which focuses on the architecture of exascale super computers. Our team focuses on the processor architecture of exascale computers and proposes a scientific processing unit which integrates a dataflow accelerator. And the results of our simulation experiments showed that dataflow computing is very efficient on high-performance computing.

《计算机学报》